9) Big O: Drop non-Dominants

```
function logItems(n) {
    for(let i=0; i<n; i++) {
        for(let j=0; j<n; j++) {          (n)^2
            console.log(i,j)
        }
    }
    for(let k=0; k<n; k++) {              (n)
        console.log(k)
    }
}
logItems(100)                          O(n^2+n)
```

→ If we took n = 100, then square would be 10,000 where the single end that added to it this equation is only 100

→ It's not really affecting the no. of operation.

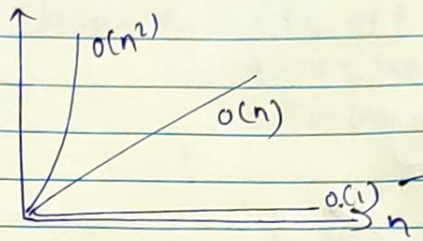
→ N squared is the dominant term and n by itself is the non-dominant term, so just remove it.

→ we drop non dominance.

10) Big O : O(1) : constant time.

```
function addItem(n) {
    return n+n          → n.
}
```

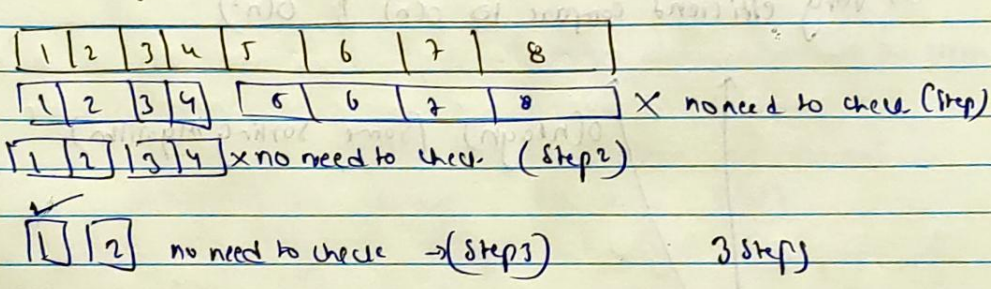
→ So a O of one a lot of time is referred to as constant time



→ No matter how much n you add it constant at the end

$n(1) - n(2) \therefore n(3) \dots n(n) \rightarrow n(1)$

II) Big O : O(logn)



→ This method is called divide & conquer.

$2^3 = 8$ (8 items in array)

$\log_2 8 = 3$

→ So if you had an array with a billion items in it, and you were going to iterate through that array

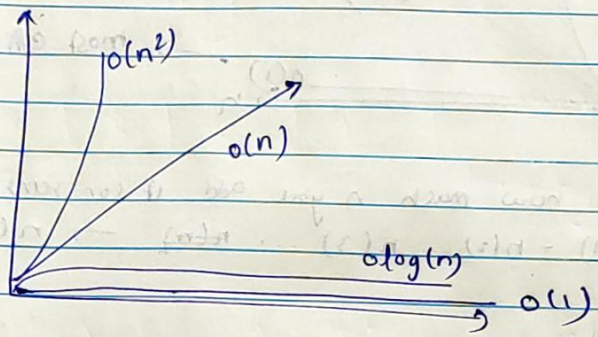
→ linearly to find something and say what you were looking for was the last item, you would have to look

→ But if you use divide & conquer you would find any item in that array in 31 steps

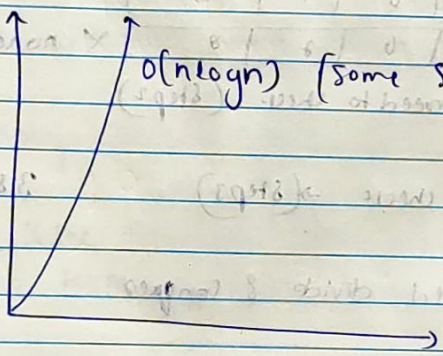
$\log_2 1073741824 = 31$

↓

O(log n)



→ very efficient compare to O(n) & O(n²)



12) Big O: Different Terms for input

```
function logTerms(a,b){
    for(let i=0; i<a; i++){          O(a)
        console.log(i)
    }                                 +
    for(let j=0; j<b; j++){
        console.log(j)               O(b)
    }
}
```

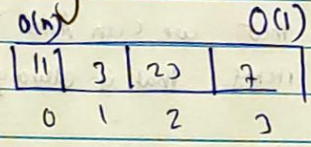
⇒ O(a+b)

→ If it is nested loop

```
function logItems(a, b) {
    for(let i=0; i<a; i++) {          O(a)
        for(let j=0; j<b; j++) {      O(b)
    } }
```

⇒ O(a*b)

13) Big O : Arrays.



myArray.push(11) → Add the item at end

myArray.pop() → remove the item from end

→ From both the operation we did not rearrange the indexes. So it is O(1)

push and pop → O(1)

* myArray.shift() → remove from front
  .unshift(11) → add the element in front
  ] O(n)

⇒ myArray.splice(1, 0, 'Hi')

1 → index position to add 'Hi'

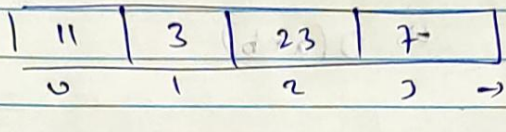
0 → no need to remove index item

11	'Hi'	3	23	22

O(n)

→ So it doesn't matter if you'r remove (or) adding somewhere in the middle of array that is O(n)

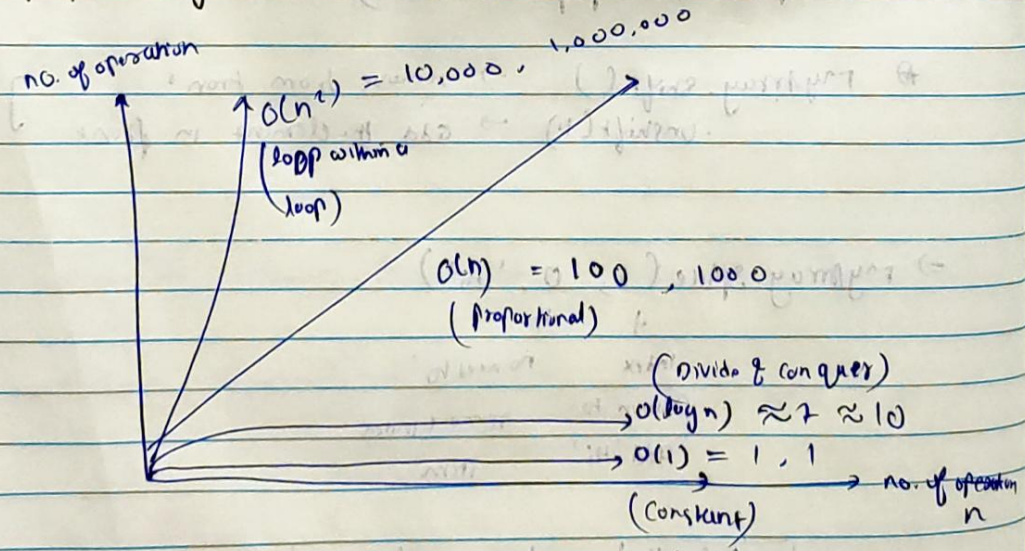
→ find the element array



→ The big advantages of array is that we can find something in an array with a million items that is always O(1)

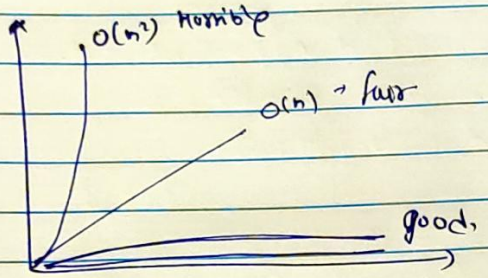
But disadvantages of array if you want add in the big beginning, because you're going to reindex it again.

14) Wrap up : Big O.



n = 100
n = 1000

→ big O cheatsheet con



→ Sort string or different ~~kind~~ kind of data other than no. the best time complexity you can get for a sorting algorithm is O(n)